Atelier 10 NextJS : Tensorflow.js et l'IA

# Introduction

Référence : <https://javascript.plainenglish.io/making-your-next-js-project-smarter-with-tensorflow-js-and-ai-a11f77515827>

L’intégration de l’IA dans le développement Web une évolution dynamique de la façon dont nous percevons et interagissons avec le Web.

Intégrer Tensorflow.js dans un projet Next.js n'est pas seulement réalisable, c'est simple. La combinaison de l’efficacité de Next.js et de la robustesse de Tensorflow.js ouvre les portes à de nombreuses possibilités.

Tensorflow.js est une bibliothèque JavaScript open source à accélération matérielle pour la formation et le déploiement de modèles d'apprentissage automatique dans le navigateur.

Les prédictions données par MobileNet sont une estimation basée sur son entraînement sur un vaste dataset. Les résultats indiquent les catégories les plus probables avec leurs niveaux de confiance respectifs.

L'intégration de la classification d'images basée sur l'apprentissage automatique dans une application web e-commerce peut améliorer de manière significative l'expérience utilisateur, l'efficacité de la gestion des produits et l'efficacité des campagnes marketing. Cela permet de proposer des recommandations plus pertinentes, de faciliter la recherche de produits et d'automatiser de nombreux processus, réduisant ainsi le travail manuel et augmentant la satisfaction des clients.

En effet, l'utilisation des prédictions d'images dans une application web e-commerce peut offrir de nombreux avantages, notamment :

1. Amélioration de l'Expérience Utilisateur :

- Recherche Visuelle : Les utilisateurs peuvent télécharger une photo d'un produit qu'ils souhaitent acheter, et le système peut automatiquement afficher des produits similaires disponibles sur le site.

Exemple : Un utilisateur voit une robe dans une publicité et veut trouver un modèle similaire sur votre site. Il télécharge l'image de la robe, et votre application utilise les prédictions pour trouver des articles visuellement similaires.

- Suggestions de Produits : Lorsque les utilisateurs naviguent sur un produit, des suggestions basées sur l'apparence visuelle des produits peuvent être proposées, améliorant ainsi la découverte de produits.

2. Gestion de l'Inventaire :

- Classification Automatique des Produits : Lors de l'ajout de nouveaux produits, les images peuvent être automatiquement classées dans les bonnes catégories, réduisant ainsi le temps et l'effort nécessaires pour organiser l'inventaire.

- Étiquetage Automatique : Les produits peuvent être automatiquement étiquetés avec des mots-clés pertinents basés sur les caractéristiques visuelles identifiées par le modèle.

Exemple : Lorsqu'un vendeur ajoute une nouvelle paire de chaussures à son catalogue, l'image est analysée et automatiquement classée sous "Chaussures > Bottes > Femmes" et étiquetée avec des mots-clés comme "cuir", "hiver", "mode".

3. Amélioration du SEO (Search Engine Optimization) :

- Balises d'Images Automatiques : Génération automatique de balises ALT pour les images des produits, ce qui peut aider à améliorer le référencement des produits sur les moteurs de recherche.

4. Personnalisation des Contenus :

- Recommandations Personnalisées : En utilisant les préférences visuelles des utilisateurs (basées sur les produits qu'ils ont visualisés ou téléchargés), le site peut offrir des recommandations de produits plus pertinentes.

Exemple : Un utilisateur cherche une lampe de bureau. Votre application utilise les prédictions pour recommander des lampes similaires en style et en forme, améliorant ainsi les chances de conversion.

5. Détection de Contenus Inappropriés :

- Filtrage Automatique : Identification et filtrage des images inappropriées ou non conformes avant qu'elles ne soient publiées sur le site.

6. Optimisation des Campagnes Marketing :

- Analyse des Tendances : En analysant les images des produits les plus populaires ou les plus vus, les équipes marketing peuvent identifier des tendances et ajuster leurs campagnes en conséquence.

- Création de Contenus : Génération automatique de descriptions de produits et de suggestions pour les campagnes publicitaires basées sur les caractéristiques visuelles des produits.

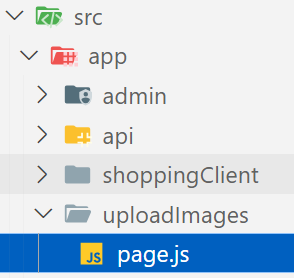
# Dépendances

Installer les packages suivants :

## npm install @tensorflow/tfjs @tensorflow-models/mobilenet

# Implémentation de la classification d'images

Créer :



Le code fera trois choses principales :

Charge le modèle MobileNet lorsque la page est initialisée.

Lorsqu'une image est téléchargée, elle appelle classifyImage qui traite et classe l'image.

Affiche la prédiction sur la page.

import React, { useState } from 'react';

import \* as mobilenet from '@tensorflow-models/mobilenet';

import \* as tf from '@tensorflow/tfjs';

export default function PageUploadImages() {

  const [img, setImg] = useState(null);

  let model;

  async function loadModel() {

    console.log("Model loading..");

    model = await mobilenet.load();

    console.log("Model loaded..");

  }

  loadModel();

  function handleImageUpload(event) {

    const image = event.target.files[0];

    classifyImage(image);

    setImg(image);

  }

  async function classifyImage(image) {

    const img = new Image();

    img.src = URL.createObjectURL(image);

    img.onload = async () => {

      const predictions = await model.classify(img);

      document.getElementById("prediction").innerHTML =

        `Predicted:<br /> ${predictions.map(p => `${p.className}: ${p.probability.toFixed(2)}`).join("<br />")}`;

    };

  }

  return (

    <div style={{ textAlign: 'center', marginTop: '4em' }}>

      <input type="file" onChange={handleImageUpload} />

      <br />

      <div>

        <p id="prediction"></p>

        <br />

        {img ? <img src={img ? URL.createObjectURL(img) : null} alt="upload-preview" style={{ maxWidth: '100%' }} /> : null}

      </div>

    </div>

  );

}

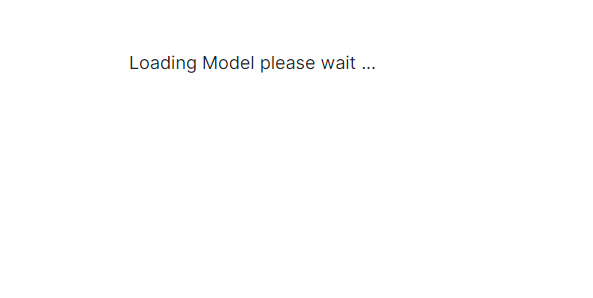
Explication :

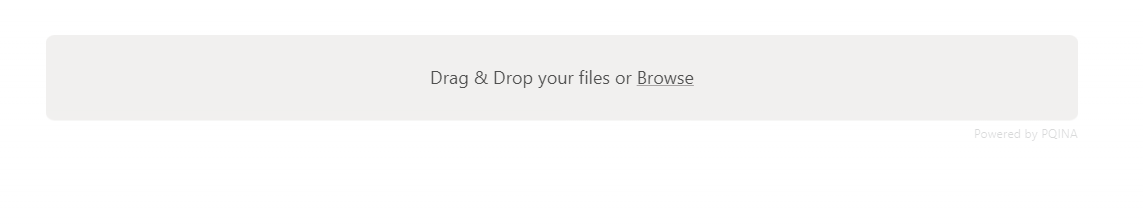
Utilisation de useEffect pour charger le modèle : Le modèle est chargé lors de la première exécution du composant grâce à useEffect, ce qui assure que le modèle est prêt avant que toute image soit classifiée.

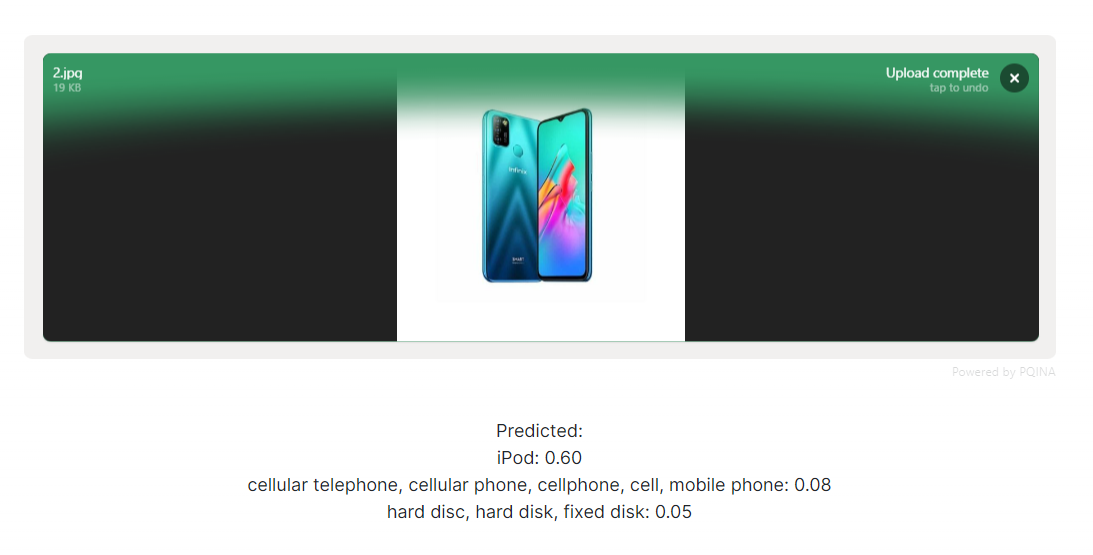
Vérification de modelLoaded avant d'appeler classifyImage : Avant d'appeler classifyImage, on vérifie si modelLoaded est true. Si ce n'est pas le cas, une erreur est loguée.

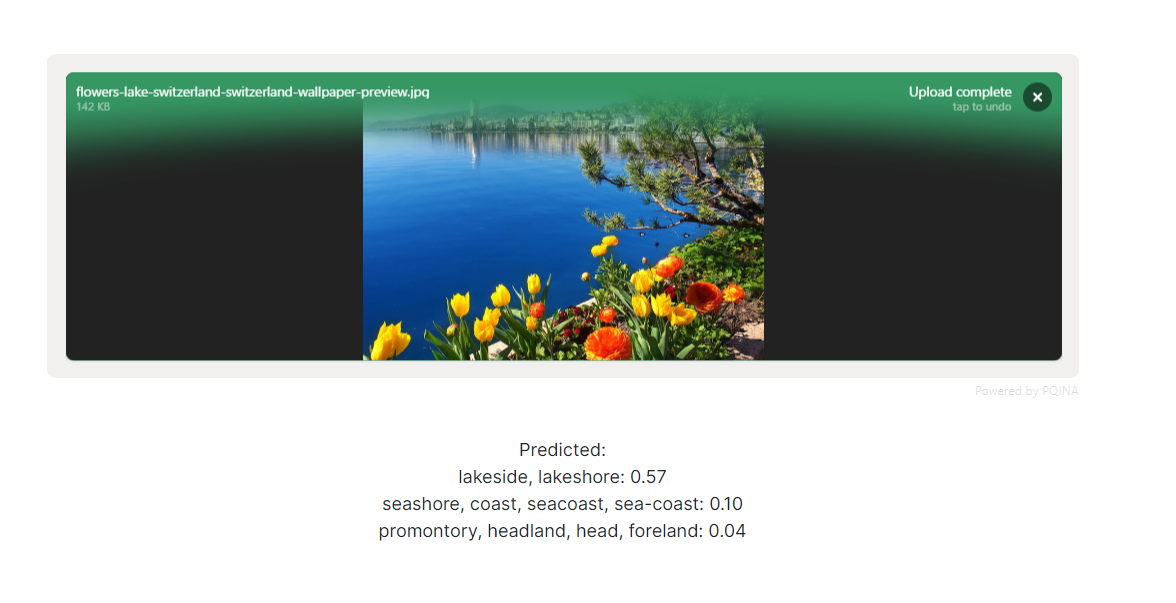
Nettoyage de l'URL de l'objet : Après l'utilisation de l'image pour la classification, l'URL de l'objet est révoquée pour libérer les ressources.

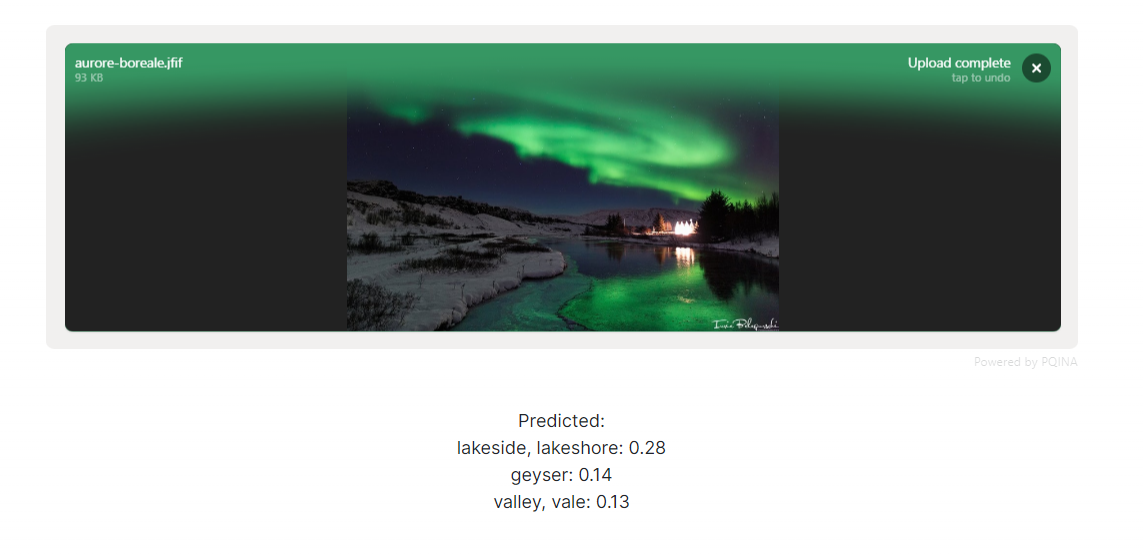
Vous pouvez tester cela avec plusieurs images. Les prédictions changeront en fonction de l'image téléchargée.











Le résultat de la classification d'image obtenu (par exemple ce dernier), produit par le modèle MobileNet, tel que "Predicted: lakeside, lakeshore: 0.28; geyser: 0.14; valley, vale: 0.13", est un modèle de réseau de neurones convolutionnel pré-entraîné sur le dataset ImageNet.

Chaque ligne correspond à une catégorie (ou classe) que le modèle a prédit comme étant possible pour l'image donnée.

Les catégories sont des noms de classes dans le dataset ImageNet, qui contient des milliers de catégories d'objets et de scènes.

lakeside, lakeshore : Ce sont des descriptions textuelles de la première catégorie prédite. Dans ce cas, cela signifie que l'image pourrait représenter un bord de lac ou une rive.

geyser : Cela signifie que l'image pourrait représenter une source d'eau qui jaillit violemment, comme celles d'Islande.

valley, vale : Cela signifie que l'image pourrait représenter une vallée ou un vallon.

0.28, 0.14, 0.13 : Ce sont les probabilités associées à chaque catégorie prédite. Ces valeurs représentent la confiance du modèle dans chaque prédiction, sous forme de pourcentages (0.28 = 28%, 0.14 = 14%, 0.13 = 13%).

Plus la probabilité est élevée, plus le modèle est confiant que l'image appartient à cette catégorie.

Les probabilités de toutes les catégories sont normalement comprises entre 0 et 1 et leur somme est égale à 1 (ou très proche de 1 à cause des erreurs d'arrondi).

Aucune des probabilités n'est très élevée (proche de 1.0), ce qui peut indiquer que l'image n'est pas très typique ou claire pour ces catégories, ou que l'image pourrait appartenir à une catégorie qui n'est pas bien représentée dans le dataset de formation du modèle.

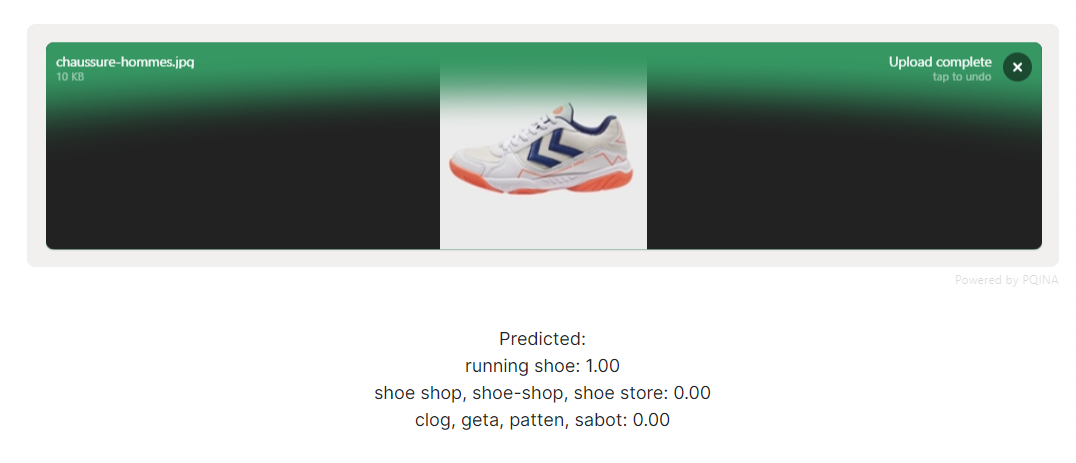
La probabilité la plus élevée étant 0.28 (28%) suggère que le modèle n'est pas très confiant dans cette prédiction particulière.

Qualité de l'image : Si l'image est floue, mal éclairée ou a un angle inhabituel, cela peut affecter les prédictions du modèle.

L'image peut contenir des éléments qui ressemblent à plusieurs catégories, ce qui peut mener à des probabilités réparties sur plusieurs classes.

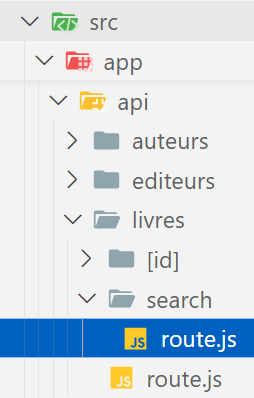
Bien que MobileNet soit puissant, il n'est pas parfait et a des limites basées sur le dataset sur lequel il a été entraîné et les classes disponibles dans ce dataset.

Voilà un exemple d’une image avec une probabilité élevée



# Recherche visuelle

Créer :



import Livre from '@/models/Livre';

import  Specialite from '@/models/Specialite';

import {  NextResponse } from 'next/server';

import connectDB from '@/lib/connectDB';

export async function GET(req) { console.log(req.query)

    try {

        await connectDB();

        let  query  = req.url;

    if (!query) {

      return NextResponse.json({ error: 'Query parameter is required' });

    }

    // Décoder les caractères spéciaux de l'URL

  query = decodeURIComponent(query);

  // Supprimer les espaces de la chaîne de requête

  query = query.replace(/\s+/g, '');

  // Extraire les mots clés de la chaîne de requête

    const searchQueries = query.split(',');

    try {

     // Recherche des spécialités correspondant aux searchQueries

    const specialites = await Specialite.find({ nomspecialite: { $in: searchQueries } }).exec();

    // Récupération des IDs des spécialités trouvées

    const specialiteIds = specialites.map(specialite => specialite.\_id);

    // Recherche des livres dont les spécialités correspondent aux IDs trouvés

    const livres = await Livre.find({ specialite: { $in: specialiteIds } }).exec();

      return NextResponse.json(livres);

    } catch (error) {

      console.error('Error fetching products:', error);

      return NextResponse.json({ error: 'Internal Server Error' });

    }

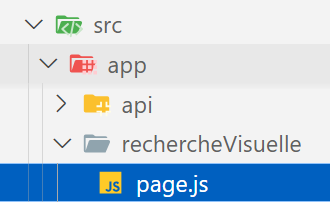
    } catch (error) {

        return NextResponse.json({ error : error });

    }

}

Puis créer :



Pour compléter l'exemple de recherche visuelle où les utilisateurs peuvent télécharger une photo d'un produit qu'ils souhaitent acheter et le système affiche automatiquement des produits similaires disponibles sur le site, vous pouvez suivre les étapes ci-dessous :

Adapter la Fonction classifyImage : Utiliser les prédictions du modèle pour effectuer une recherche dans votre base de données de produits.

Intégrer une Recherche de Produits Similaires : Ajouter une fonction pour rechercher des produits similaires dans votre inventaire basé sur les résultats de classification.

Afficher les Résultats : Afficher les produits similaires sur l'interface utilisateur.

'use client'

import React, { useState, useEffect } from 'react';

import \* as mobilenet from '@tensorflow-models/mobilenet';

import \* as tf from '@tensorflow/tfjs';

import axios from 'axios';

import { FilePond, registerPlugin } from 'react-filepond';

import 'filepond/dist/filepond.min.css';

import FilePondPluginImageExifOrientation from 'filepond-plugin-image-exif-orientation';

import FilePondPluginImagePreview from 'filepond-plugin-image-preview';

import 'filepond-plugin-image-preview/dist/filepond-plugin-image-preview.css';

registerPlugin(FilePondPluginImageExifOrientation, FilePondPluginImagePreview);

export default function PageUploadImages() {

  const [img, setImg] = useState(null);

  const [files, setFiles] = useState([]);

  const [model, setModel] = useState(null);

  const [modelLoaded, setModelLoaded] = useState(false);

  const [similarProducts, setSimilarProducts] = useState([]);

  useEffect(() => {

    async function loadModel() {

      console.log("Model loading...");

      const loadedModel = await mobilenet.load();

      setModel(loadedModel);

      setModelLoaded(true);

      console.log("Model loaded.");

    }

    loadModel();

  }, []);

  const serverOptions = () => {

    return {

      process: (fieldName, file, metadata, load, error, progress, abort) => {

        const data = new FormData();

        data.append('file', file);

        data.append('upload\_preset', 'Ecommerce\_cloudinary');

        data.append('cloud\_name', 'iset-sfax');

        data.append('public\_id', file.name);

        axios.post('https://api.cloudinary.com/v1\_1/iset-sfax/image/upload', data)

          .then((response) => response.data)

          .then((data) => {

            setImg(data.url);

            if (modelLoaded) {

              classifyImage(file); // Pass the file directly here

            } else {

              console.error('Model not loaded yet.');

            }

            load(data);

          })

          .catch((error) => {

            console.error('Error uploading file:', error);

            error('Upload failed');

            abort();

          });

      },

    };

  };

  async function classifyImage(file) {

    if (!modelLoaded) {

      console.error('Model not loaded yet.');

      return;

    }

    const img = new Image();

    const url = URL.createObjectURL(file);

    img.src = url;

    img.onload = async () => {

      const predictions = await model.classify(img);

      document.getElementById("prediction").innerHTML =

        `Predicted:<br /> ${predictions.map(p => `${p.className}: ${p.probability.toFixed(2)}`).join("<br />")}`;

      URL.revokeObjectURL(url); // Clean up the object URL after using it

      // Fetch similar products based on predictions

      fetchSimilarProducts(predictions);

    };

  }

  async function fetchSimilarProducts(predictions) {

    try {

      const searchQueries = predictions.map(p => p.className).join(',');

      const response = await axios.get(`http://localhost:3000/api/livres/search?query=${searchQueries}`);

      console.log(response.data)

      setSimilarProducts(response.data);

    } catch (error) {

      console.error('Error fetching similar products:', error);

    }

  }

  return (

    <div style={{ textAlign: 'center', marginTop: '4em' }}>

      <div style={{ width: "80%", margin: "auto", padding: "1%" }}>

      { modelLoaded ?

      <FilePond

      files={files}

      acceptedFileTypes="image/\*"

      onupdatefiles={setFiles}

      allowMultiple={false}

      server={serverOptions()}

      name="file"

    />

     : "Loading Model please wait ..."}

      </div>

      <br />

      <div>

        <p id="prediction"></p>

        {similarProducts.length > 0 && (

          <div>

            <h2>Similar Products</h2>

            <ul>

              {similarProducts.map(product => (

                <li key={product.\_id}>

                  <img src={product.couverture} alt={product.titre} style={{width: '100px', height: '100px'}}/>

                  {product.titre}

                </li>

              ))}

            </ul>

          </div>

        )}

      </div>

    </div>

  );

}

Explication :

1. **Chargement du Modèle** : Le modèle MobileNet est chargé au démarrage du composant.
2. **Téléchargement et Traitement des Images** : Les utilisateurs peuvent télécharger une image via FilePond. L'image est ensuite téléchargée sur Cloudinary et classifiée par MobileNet.
3. **Recherche de Produits Similaires** : Les prédictions du modèle sont utilisées pour rechercher des produits similaires dans la base de données.
4. **Affichage des Résultats** : Les produits similaires sont affichés sous les prédictions.

